2 Hash functions

This chapter covers

- Hash functions and their security properties
- The widely adopted hash functions in use today
- Other types of hashing that exist

Attributing global unique identifiers to anything, that's the promise of the first cryptographic construction you'll learn about in this chapter—the *hash function*. Hash functions are everywhere in cryptography—everywhere! Informally, they take as input any data you'd like and produce a unique string of bytes in return. Given the same input, the hash function always reproduces the same string of bytes. This might seem like nothing, but this simple fabrication is extremely useful to build many other constructions in cryptography. In this chapter, you will learn everything there is to know about hash functions and why they are so versatile.

2.1 What is a hash function?

In front of you, a download button is taking a good chunk of the page. You can read the letters *DOWNLOAD*, and clicking this seems to redirect you to a different website containing a file. Below it, lies a long string of unintelligible letters:

```
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

It is followed by what looks like an acronym of some sort: `sha256sum`. Sound familiar? You've probably downloaded something in your past life that was also accompanied with such an odd string (figure 2.1).

Figure 2.1 A web page linking to an external website containing a file. The external website cannot modify the content of the file because the first page provides a hash or digest of the file, which ensures the integrity over the downloaded file.

If you've ever wondered what was to be done with that long string:

1. Click the button to download the file
2. Use the SHA-256 algorithm to *hash* the downloaded file
3. Compare the output (the digest) with the long string displayed on the web page

This allows you to verify that you downloaded the right file.

> **NOTE** The output of a hash function is often called a *digest* or a *hash*. I use the two words interchangeably throughout this book. Others might call it a *checksum* or a *sum*, which I avoid, as those terms are primarily used by noncryptographic hash functions and could lead to more confusion. Just keep that in mind when different codebases or documents use different terms.

To try hashing something, you can use the popular OpenSSL library. It offers a multipurpose command-line interface (CLI) that comes by default in a number of systems including macOS. For example, this can be done by opening the terminal and writing the following line:

```
$ openssl dgst -sha256 downloaded_file
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

With the command, we used the SHA-256 hash function to transform the input (the downloaded file) into a unique identifier (the value echoed by the command). What do these extra steps provide? They provide *integrity and authenticity*. It tells you that what you downloaded is indeed the file you were meant to download.

All of this works, thanks to a security property of the hash function called *second preimage resistance*. This math-inspired term means that from the long output of the hash

function, `f63e...`, you cannot find another file that will hash to the same output, `f63e....` In practice, it means that this digest is closely tied to the file you're downloading and that no attacker should be able to fool you by giving you a different file.

The hexadecimal notation

By the way, the long output string `f63e...` represents binary data displayed in *hexadecimal* (a base-16 encoding, using numbers from 0 to 9 and letters from *a* to *f* to represent several bits of data). We could have displayed the binary data with 0s and 1s (base 2), but it would have taken more space. Instead, the hexadecimal encoding allows us to write 2 alphanumeric characters for every 8 bits (1 byte) encountered. It is somewhat readable by humans and takes less space. There are other ways to encode binary data for human consumption, but the two most widely used encodings are hexadecimal and base64. The larger the base, the less space it takes to display a binary string, but at some point, we run out of human-readable characters.

Note that this long digest is controlled by the owner(s) of the web page, and it could easily be replaced by anyone who can modify the web page. (If you are not convinced, take a moment to think about it.) This means that we need to trust the page that gave us the digest, its owners, and the mechanism used to retrieve the page (while we don't need to trust the page that gave us the file we downloaded). In this sense, *the hash function alone does not provide integrity*. The integrity and authenticity of the downloaded file comes from the digest combined with the trusted mechanism that gave us the digest (HTTPS in this case). We will talk about HTTPS in chapter 9, but for now, imagine that it magically allows you to communicate securely with a website.

Back to our hash function, which can be visualized as the black box in figure 2.2. Our black box takes a single input and gives out a single output.

Figure 2.2 A hash function takes an arbitrary-length input (a file, a message, a video, and so on) and produces a fixed-length output (for example, 256 bits for SHA-256). Hashing the same input produces the same digest or hash.

The *input* of this function can be of any size. It can even be empty. The *output* is always of the same length and *deterministic*: it always produces the same result if given the same input. In our example, SHA-256 always provides an output of 256 bits (32 bytes), which is always encoded as 64 alphanumeric characters in hexadecimal. One major property of a hash function is that one cannot revert the algorithm, meaning that one shouldn't be able to find the input from just the output. We say that hash functions are *one-way*.

To illustrate how a hash function works in practice, we'll hash different inputs with the SHA-256 hash function using the same OpenSSL CLI. The following terminal session shows this.

```
$ echo -n "hello" | openssl dgst -sha256
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
$ echo -n "hello" | openssl dgst -sha256
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
$ echo -n "hella" | openssl dgst -sha256
70de66401b1399d79b843521ee726dcec1e9a8cb5708ec1520f1f3bb4b1dd984
$ echo -n "this is a very very very very very very
  ➥ very very very long sentence" | openssl dgst -sha256
1166e94d8c45fd8b269ae9451c51547dddec4fc09a91f15a9e27b14afee30006
```

Hashing the same input produces the same result.

A tiny change in the input completely changes the output.

The output is always of the same size, no matter the input size.

In the next section, we will see what are the exact security properties of hash functions.

2.2 Security properties of a hash function

Hash functions in applied cryptography are constructions that were commonly defined to provide three specific security properties. This definition has changed over time as we will see in the next sections. But for now, let's define the three strong foundations that make up a hash function. This is important as you need to understand where hash functions can be useful and where they will not work.

The first one is *pre-image resistance*. This property ensures that no one should be able to reverse the hash function in order to recover the input given an output. In figure 2.3, we illustrate this "one-wayness" by imagining that our hash function is like a blender, making it impossible to recover the ingredients from the produced smoothie.

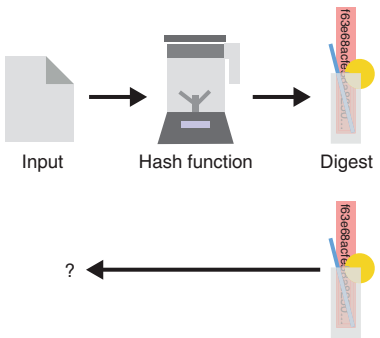

Figure 2.3 Given the digest produced by a hash function (represented as a blender here), it is impossible (or technically so hard we assume it will never happen) to reverse it and find the original input used. This security property is called *pre-image resistance*.

WARNING Is this true if your input is small? Let's say that it's either *oui* or *non*, then it is easy for someone to hash all the possible 3-letter words and find out what the input was. What if your input space is small? Meaning that you always hash variants of the sentence, "I will be home on Monday at 3 a.m.," for example. Here, one who can predict this but does not know exactly the day of the

week or the hour can still hash all possible sentences until it produces the correct output. As such, this first pre-image security property has an obvious caveat: *you can't hide something that is too small or that is predictable.*

The second property is *second pre-image resistance*. We already saw this security property when we wanted to protect the integrity of a file. The property says the following: if I give you an input and the digest it hashes to, you should not be able to find a different input that hashes to the same digest. Figure 2.4 illustrates this principle.

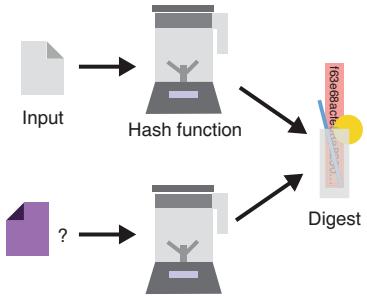

Figure 2.4 Considering an input and its associated digest, one should never be able to find a different input that hashes to the same output. This security property is called *second pre-image resistance*.

Note that *we do not control the first input*. This emphasis is important to understand the next security property for hash functions.

Finally, the third property is *collision resistance*. It guarantees that no one should be able to produce two different inputs that hash to the same output (as seen in figure 2.5). Here an attacker can choose the two inputs, unlike the previous property that fixes one of the inputs.

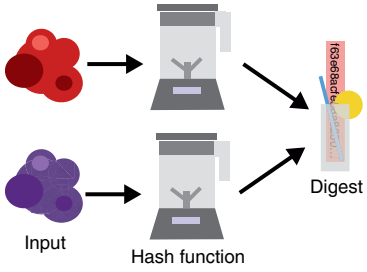

Figure 2.5 One should never be able to find two inputs (represented on the left as two random blobs of data) that hash to the same output value (on the right). This security property is called *collision resistance*.

People often confuse collision resistance and second pre-image resistance. Take a moment to understand the differences.

The random oracle

In addition, hash functions are usually designed so that their digests are *unpredictable and random*. This is useful because one cannot always prove a protocol to be

(continued)

secure, thanks to one of the security properties of a hash function we talked about (like collision resistance, for example). Many protocols are instead proven in the *random oracle model*, where a fictional and ideal participant called a random oracle is used. In this type of protocol, one can send any inputs as requests to that random oracle, which is said to return completely random outputs in response, and like a hash function, giving it the same input twice returns the same output twice.

Proofs in this model are sometimes controversial as we don't know for sure if we can replace these random oracles with real hash functions (in practice). Yet, many legitimate protocols are proven secure using this method, where hash functions are seen as more ideal than they probably are.

2.3 Security considerations for hash functions

So far, we saw three security properties of a hash function:

- Pre-image resistance
- Second pre-image resistance
- Collision resistance

These security properties are often meaningless on their own; it all depends on how you make use of the hash function. Nonetheless, it is important that we understand some limitations here before we look at some of the real-world hash functions.

First, these security properties assume that you are (reasonably) using the hash function. Imagine that I either hash the word *yes* or the word *no*, and I then publish the digest. If you have some idea of what I am doing, you can simply hash both of the words and compare the result with what I give you. Because there are no secrets involved, and because the hashing algorithm we used is public, you are free to do that. And indeed, one could think this would break the pre-image resistance of the hash function, but I'll argue that your input was not "random" enough. Furthermore, because a hash function accepts an arbitrary-length input and always produces an output of the same length, there are also an infinite number of inputs that hash to the same output. Again, you could say, "Well, isn't this breaking the second pre-image resistance?" Second pre-image resistance is merely saying that it is extremely hard to find another input, so hard we assume it's in practice impossible but not theoretically impossible.

Second, the size of the digest *does* matter. This is not a peculiarity of hash functions by any means. All cryptographic algorithms must care about the size of their parameters in practice. Let's imagine the following extreme example. We have a hash function that produces outputs of length 2 bits in a uniformly random fashion (meaning that it will output `00` 25% of the time, `01` 25% of the time, and so on). You're not going to have to do too much work to produce a collision: after hashing a few random input strings, you should be able to find two that hash to the same output. For this rea-

son, there is a *minimum output size* that a hash function *must* produce in practice: 256 bits (or 32 bytes). With this large an output, collisions should be out of reach unless a breakthrough happens in computing.

How was this number obtained? In real-world cryptography, algorithms aim for a minimum of 128 bits of security. It means that an attacker who wants to break an algorithm (providing 128-bit security) has to perform around 2^{128} operations (for example, trying all the possible input strings of length 128 bits would take 2^{128} operations). For a hash function to provide all three security properties mentioned earlier, it needs to provide at least 128 bits of security against all three attacks. The easiest attack is usually to find collisions due to the *birthday bound*.

The birthday bound

The birthday bound takes its roots from probability theory in which the birthday problem reveals some unintuitive results. How many people do you need in a room so that with at least a 50% chance, two people share the same birthday (that's a collision). It turns out that 23 people taken at random are enough to reach these odds! Weird right?

This is called the *birthday paradox*. In practice, when we randomly generate strings from a space of 2^N possibilities, you can expect with a 50% chance that someone will find a collision after having generated approximately $2^{N/2}$ strings.

If our hash function generates random outputs of 256 bits, the space of all outputs is of size 2^{256}. This means that collisions can be found with good probability after generating 2^{128} digests (due to the birthday bound). This is the number we're aiming for, and this is why hash functions at a minimum must provide 256-bit outputs.

Certain constraints sometimes push developers to reduce the size of a digest by *truncating it* (removing some of its bytes). In theory, this is possible but can greatly reduce security. In order to achieve 128-bit security at a minimum, a digest must not be truncated under:

- 256 bits for collision resistance
- 128 bits for pre-image and second pre-image resistance

This means that depending on what property one relies on, the output of a hash function can be truncated to obtain a shorter digest.

2.4 Hash functions in practice

As we said earlier, in practice, hash functions are rarely used alone. They are most often combined with other elements to either create a cryptographic primitive or a cryptographic protocol. We will look at many examples of using hash functions to build more complex objects in this book, but this section describes a few different ways hash functions have been used in the real world.

2.4.1 Commitments

Imagine that you know that a stock in the market will increase in value and reach $50 in the coming month, but you really can't tell your friends about it (for some legal reason perhaps). You still want to be able to tell your friends that you knew about it after the fact because you're smug (don't deny it). What you can do is to commit to a sentence like, "Stock *X* will reach $50 next month." To do this, hash the sentence and give your friends the output. A month later, reveal the sentence. Your friends will be able to hash the sentence to observe that indeed, it is producing the same output.

This is what we call a *commitment scheme*. Commitments in cryptography generally try to achieve two properties:

- *Hiding*—A commitment must hide the underlying value.
- *Binding*—A commitment must hide a single value. In other words, if you commit to a value *x*, you shouldn't be able to later successfully reveal a different value *y*.

Exercise

Can you tell if a hash function provides hiding and binding if used as a commitment scheme?

2.4.2 Subresource integrity

It happens (often) that web pages import external JavaScript files. For example, a lot of websites use Content Delivery Networks (CDNs) to import JavaScript libraries or web-framework-related files in their pages. Such CDNs are placed in strategic locations in order to quickly deliver these files to visitors. Yet, if the CDN goes rogue and decides to serve malicious JavaScript files, this could be a real issue. To counter this, web pages can use a feature called *subresource integrity* that allows the inclusion of a digest in the import tag:

```
<script src="https://code.jquery.com/jquery-2.1.4.min.js"
   integrity="sha256-8WqyJLuWKRBVhxXIL1jBDD7SDxU936oZkCnxQbWwJVw="></script>
```

This is exactly the same scenario we talked about in the introduction of this chapter. Once the JavaScript file is retrieved, the browser hashes it (using SHA-256) and verifies that it corresponds to the digest that was hardcoded in the page. If it checks out, the JavaScript file gets executed as its integrity has been verified.

2.4.3 BitTorrent

Users (called *peers*) around the world use the BitTorrent protocol to share files directly among each other (what we also call *peer-to-peer*). To distribute a file, it is cut into chunks and each chunk is individually hashed. These hashes are then shared as a source of trust to represent the file to download.

BitTorrent has several mechanisms to allow a peer to obtain the different chunks of a file from different peers. In the end, the integrity of the entire file is verified by hashing each of the downloaded chunks and matching the output to its respectively known digests (before reassembling the file from the chunks). For example, the following "magnet link" represents the Ubuntu operating system, v19.04. It is a digest (represented in hexadecimal) obtained from hashing the metadata about the file as well as all the chunks' digests.

```
magnet:?xt=urn:btih:b7b0fbab74a85d4ac170662c645982a862826455
```

2.4.4 Tor

The Tor browser's goal is to give individuals the ability to browse the internet anonymously. Another feature is that one can create hidden web pages, whose physical locations are difficult to track. Connections to these pages are secured via a protocol that uses the web page's public key. (We will see more about how that works in chapter 9 when we talk about session encryption.) For example, Silk Road, which used to be the eBay of drugs until it got seized by the FBI, was accessible via `silkroad6ownowfk.onion` in the Tor browser. This base32 string actually represented the hash of Silk Road's public key. Thus, by knowing the onion address, you can authenticate the public key of the hidden web page you're visiting and be sure that you're talking to the right page (and not an impersonator). If this is not clear, don't worry, I'll mention this again in chapter 9.

> **Exercise**
> By the way, there is no way this string represents 256 bits (32 bytes), right? How is this secure then, according to what you learned in section 2.3? Also, can you guess how the Dread Pirate Roberts (the pseudonym of Silk Road's webmaster) managed to obtain a hash that contains the name of the website?

In all examples in this section, a hash function provided *content integrity* or *authenticity* in situations where:

- Someone might tamper with the content being hashed.
- The hash is securely communicated to you.

We sometimes also say that we *authenticate* something or someone. It is important to understand that if the hash is not obtained securely, then anyone can replace it with the hash of something else! Thus, it does not provide integrity by itself. The next chapter on message authentication code will fix this by introducing *secrets*. Let's now look at what actual hash function algorithms you can use.

2.5 Standardized hash functions

We mentioned SHA-256 in our previous example, which is only one of the hash functions we can use. Before we go ahead and list the recommended hash functions of our time, let's first mention other algorithms that people use in real-world applications that are not considered cryptographic hash functions.

First, functions like CRC32 are *not* cryptographic hash functions but error-detecting code functions. While they helpfully detect some simple errors, they provide none of the previously mentioned security properties and are not to be confused with the hash functions we are talking about (even though they might share the name sometimes). Their output is usually referred to as a *checksum*.

Second, popular hash functions like MD5 and SHA-1 are considered broken nowadays. While they were both the standardized and widely accepted hash functions of the 1990s, MD5 and SHA-1 were shown to be broken in 2004 and 2016, respectively, when collisions were published by different research teams. These attacks were successful partly because of advances in computing, but mostly because flaws were found in the way the hash functions were designed.

Deprecation is hard

Both MD5 and SHA-1 were considered good hash functions until researchers demonstrated their lack of resistance from collisions. It remains that today, their pre-image and second pre-image resistance have not been affected by any attack. This does not matter for us as we want to only talk about secure algorithms in this book. Nonetheless, you will still see people using MD5 and SHA-1 in systems that only rely on the pre-image resistance of these algorithms and not on their collision resistance. These offenders often argue that they cannot upgrade the hash functions to more secure ones because of legacy and backward compatibility reasons. As the book is meant to last in time and be a beam of bright light for the future of real-world cryptography, this will be the last time I mention these hash functions.

The next two sections introduce SHA-2 and SHA-3, which are the two most widely used hash functions. Figure 2.6 introduces these functions.

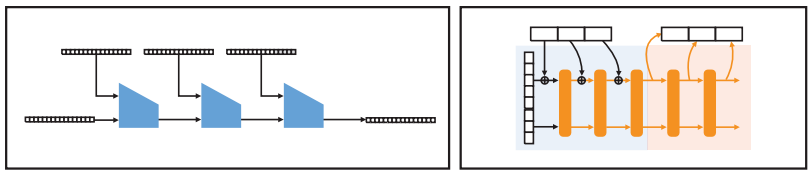

Figure 2.6 SHA-2 and SHA-3, the two most widely adopted hash functions. SHA-2 is based on the Merkle–Damgård construction, while SHA-3 is based on the sponge construction.

2.5.1 The SHA-2 hash function

Now that we have seen what hash functions are and had a glimpse at their potential use cases, it remains to be seen which hash functions we can use in practice. In the next two sections, I introduce two widely accepted hash functions, and I also give high-level explanations of how they work from the inside. The high-level explanations should not provide deeper insights on how to use hash functions because the black box descriptions I gave should be enough. But nevertheless, it is interesting to see how these cryptographic primitives were designed by cryptographers in the first place.

The most widely adopted hash function is the *Secure Hash Algorithm 2* (SHA-2). SHA-2 was invented by NSA and standardized by NIST in 2001. It was meant to add itself to the aging Secure Hash Algorithm 1 (SHA-1) already standardized by NIST. SHA-2 provides 4 different versions, producing outputs of 224, 256, 384, or 512 bits. Their respective names omit the version of the algorithm: SHA-224, SHA-256, SHA-384, and SHA-512. In addition, two other versions, SHA-512/224 and SHA-512/256, provide 224-bit and 256-bit output, respectively, by truncating the result of SHA-512.

In the following terminal session, we call each variant of SHA-2 with the OpenSSL CLI. Observe that calling the different variants with the same input produces outputs of the specified lengths that are completely different.

```
$ echo -n "hello world" | openssl dgst -sha224
2f05477fc24bb4faefd86517156dafdecec45b8ad3cf2522a563582b
$ echo -n "hello world" | openssl dgst -sha256
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
$ echo -n "hello world" | openssl dgst -sha384
fdbd8e75a67f29f701a4e040385e2e23986303ea10239211af907fcbb83578b3
  ➥ e417cb71ce646efd0819dd8c088de1bd
$ echo -n "hello world" | openssl dgst -sha512
309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f
  ➥ 989dd35bc5ff499670da34255b45b0cfd830e81f605dcf7dc5542e93ae9cd76f
```

Nowadays, people mostly use SHA-256, which provides the minimum 128 bits of security needed for our three security properties, while more paranoid applications make use of SHA-512. Now, let's look at a simplified explanation of how SHA-2 works.

The Exclusive OR operation

To understand what follows, you need to understand the *XOR* (exclusive OR) operation. XOR is a bitwise operation, meaning that it operates on bits. The following figure shows how this works. XOR is ubiquitous in cryptography, so make you sure you remember it.

XOR

$$\begin{aligned} 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \\ 0 \oplus 1 &= 1 \\ 0 \oplus 0 &= 0 \end{aligned}$$

(continued)

Exclusive OR or XOR (often denoted as ⊕) operates on 2 bits. It is similar to the OR operation except for the case where both operands are 1s.

It all starts with a special function called a *compression function*. A compression function takes two inputs of some size and produces one output of the size of one of the inputs. Put simply, it takes some data and returns less data. Figure 2.7 illustrates this.

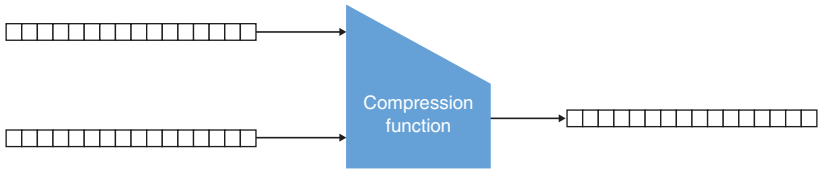

Figure 2.7 A compression function takes two different inputs of size *X* and *Y* (here both 16 bytes) and returns an output of size either *X* or *Y*.

While there are different ways of building a compression function, SHA-2 uses the *Davies–Meyer* method (see figure 2.8), which relies on a *block cipher* (a cipher that can encrypt a fixed-size block of data). I mentioned the AES block cipher in chapter 1, but you haven't yet learned about it. For now, accept the compression function as a black box until you read chapter 4 on authenticated encryption.

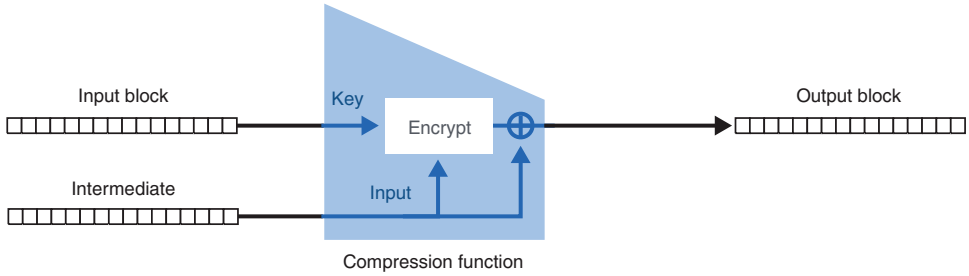

Figure 2.8 An illustration of a compression function built via the Davies–Meyer construction. The compression function's first input (the *input block*) is used as the key to a block cipher. The second input (the *intermediate value*) is used as input to be encrypted by the block cipher. It is then used again by XORing itself with the output of the block cipher.

SHA-2 is a *Merkle–Damgård* construction, which is an algorithm (invented by Ralph Merkle and Ivan Damgård independently) that hashes a message by iteratively calling such a compression function. Specifically, it works by going through the following two steps.

First, it applies a *padding* to the input we want to hash, then cuts the input into blocks that can fit into the compression function. Padding means to append specific bytes to the input in order to make its length a multiple of some block size. Cutting the padded input into chunks of the same block size allows us to fit these in the first argument of the compression function. For example, SHA-256 has a block size of 512 bit. Figure 2.9 illustrates this step.

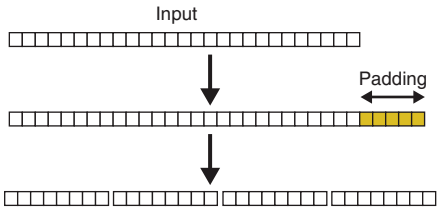

Figure 2.9 The first step of the Merkle–Damgård construction is to add some padding to the input message. After this step, the input length should be a multiple of the input size of the compression function in use (for example, 8 bytes). To do this, we add 5 bytes of padding at the end to make it 32 bytes. We then cut the messages into 4 blocks of 8 bytes.

Second, it iteratively applies the compression function to the message blocks, using the previous output of the compression function as second argument to the compression function. The final output is the *digest*. Figure 2.10 illustrates this step.

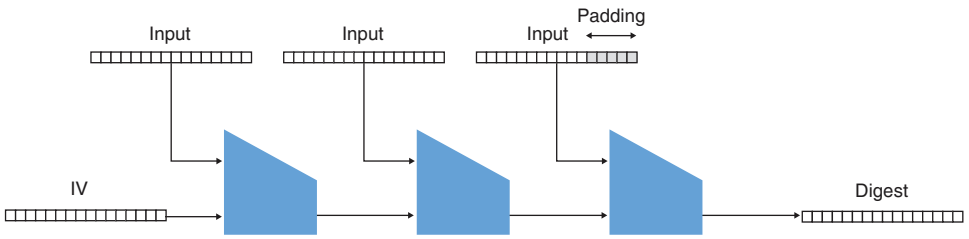

Figure 2.10 The Merkle–Damgård construction iteratively applies a compression function to each block of the input to be hashed and the output of the previous compression function. The final call to the compression function directly returns the digest.

And this is how SHA-2 works, by iteratively calling its compression function on fragments of the input until everything is processed into a final digest.

> **NOTE** The Merkle–Damgård construction is proven collision resistant if the compression function itself is. Thus, the security of the *arbitrary-length input* hash function is reduced to the security of a *fixed-sized* compression function, which is easier to design and analyze. Therein lies the ingenuity of the Merkle–Damgård construction.

In the beginning, the second argument to the compression function is usually fixed and standardized to be a "nothing-up-my-sleeve" value. Specifically, SHA-256 uses the square roots of the first prime numbers to derive this value. A nothing-up-my-sleeve

value is meant to convince the cryptographic community that it was not chosen to make the hash function weaker (for example, in order to create a backdoor). This is a popular concept in cryptography.

> **WARNING** While SHA-2 is a perfectly fine hash function to use, it is not suitable for hashing secrets. This is because of a downside of the Merkle–Damgård construction, which makes SHA-2 vulnerable to an attack (called a *length-extension attack*) if used to hash secrets. We will talk about this in more detail in the next chapter.

2.5.2 The SHA-3 hash function

As I mentioned earlier, both the MD5 and SHA-1 hash functions were broken somewhat recently. These two functions made use of the same Merkle–Damgård construction I described in the previous section. Because of this, and the fact that SHA-2 is vulnerable to length-extension attacks, NIST decided in 2007 to organize an open competition for a new standard: *SHA-3*. This section introduces the newer standard and attempts to give a high-level explanation of its inner workings.

In 2007, 64 different candidates from different international research teams entered the SHA-3 contest. Five years later, Keccak, one of the submissions, was nominated as the winner and took the name SHA-3. In 2015, SHA-3 was standardized in the FIPS Publication 202 (https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf).

SHA-3 observes the three previous security properties we talked about and provides as much security as the SHA-2 variants. In addition, it is not vulnerable to length-extension attacks and can be used to hash secrets. For this reason, it is now the recommended hash function to use. It offers the same variants as SHA-2, this time indicating the full name SHA-3 in their named variants: SHA-3-224, SHA-3-256, SHA-3-384, and SHA-3-512. Thus, similarly to SHA-2, SHA-3-256 provides 256 bits of output, for example. Let me now take a few pages to explain how SHA-3 works.

SHA-3 is a cryptographic algorithm built on top of a *permutation*. The easiest way to understand a permutation is to imagine the following: you have a set of elements on the left and the same set of elements on the right. Now trace arrows going from each element on the left to the right. Each element can only have one arrow starting from and terminating to it. You now have one permutation. Figure 2.11 illustrates this principle. By definition, any permutation is also *reversible*, meaning that from the output we can find the input.

SHA-3 is built with a *sponge construction*, a different construction from Merkle–Damgård that was invented as part of the SHA-3 competition. It is based on a particular permutation called *keccak-f* that takes an input and returns an output of the same size.

> **NOTE** We won't explain how keccak-f was designed, but you will get an idea in chapter 4 about this because it substantially resembles the AES algorithm (with the exception that it doesn't have a key). This is no accident, as one of the inventors of AES was also one of the inventors of SHA-3.

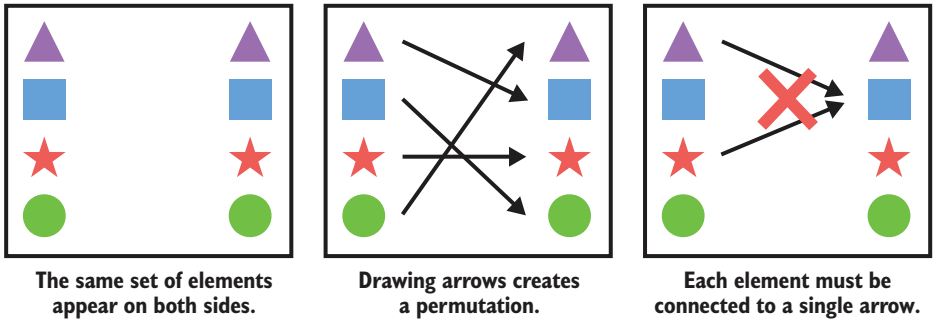

Figure 2.11 An example permutation acting on four different shapes. You can use the permutation described by the arrows in the middle picture to transform a given shape.

In the next few pages, I use an 8-bit permutation to illustrate how the sponge construction works. Because the permutation is set in stone, you can imagine that figure 2.12 is a good illustration of the mapping created by this permutation on all possible 8-bit inputs. Compared to our previous explanation of a permutation, you can also imagine that each possible 8-bit string is what we represented as different shapes (`000...` is a triangle, `100...` is a square, and so on).

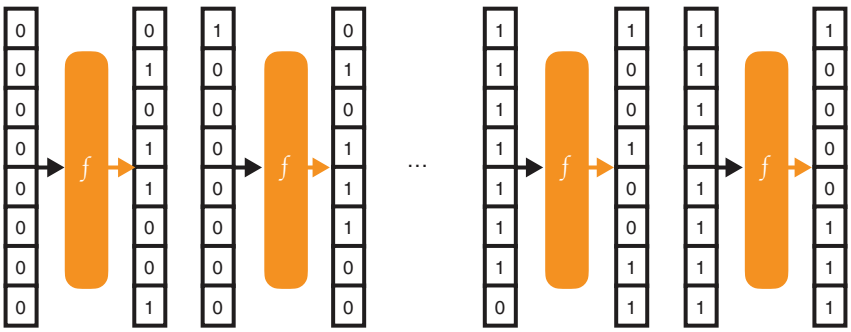

Figure 2.12 A sponge construction makes use of a specified permutation *f*. By operating on an input, our example permutation creates a mapping between all possible input of 8 bits and all possible output of 8 bits.

To use a permutation in our sponge construction, we also need to define an arbitrary division of the input and the output into a *rate* and a *capacity*. It's a bit weird but stick with it. Figure 2.13 illustrates this process.

Where we set the limit between the rate and the capacity is arbitrary. Different versions of SHA-3 use different parameters. We informally point out that the capacity is to be treated like a secret, and the larger it is, the more secure the sponge construction.

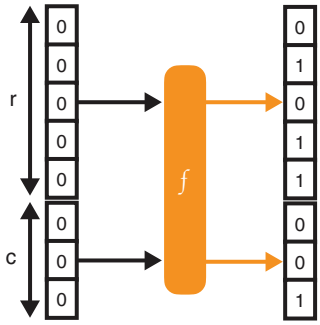

Figure 2.13 The permutation *f* randomizes an input of size 8 bits into an output of the same size. In a sponge construction, this permutation's input and output are divided into two parts: the rate (of size *r* and the capacity (of size *c*).

Now, like all good hash functions, we need to be able to hash something, right? Otherwise, it's a bit useless. To do that, we simply XOR (⊕) the input with the rate of the permutation's input. In the beginning, this is just a bunch of 0s. As we pointed out earlier, the capacity is treated like a secret, so we won't XOR anything with it. Figure 2.14 illustrates this.

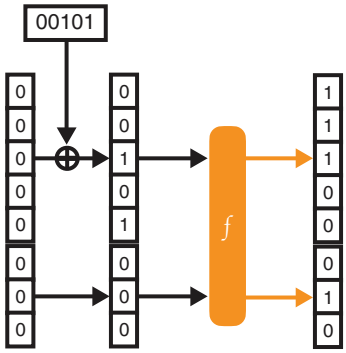

Figure 2.14 To absorb the 5 bits of input `00101`, a sponge construction with a rate of 5 bits can simply XOR the 5 bits with the rate (which is initialized to 0s). The permutation then randomizes the state.

The output obtained should now look random (although we can trivially find what the input is as a permutation is reversible by definition). What if we want to ingest a larger input? Well, similarly to what we did with SHA-2, we would

1. Pad the input if necessary, then divide the input into blocks of the rate size.
2. Iteratively call the permutation while XORing each block with the input of a permutation and permuting the *state* (the intermediate value output by the last operation) after each block has been XORed.

I ignore the padding in the rest of these explanations for the sake of simplification, but padding is an important step of the process to distinguish between inputs like `0` and `00`, for example. Figure 2.15 pictures these two steps.

So far so good, but we still haven't produced a digest. To do this, we can simply use the rate of the last state of the sponge (again, we are not touching the capacity). To

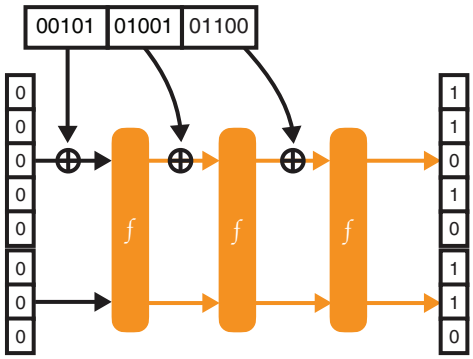

Figure 2.15 In order to absorb inputs larger than the rate size, a sponge construction iteratively XORs input blocks with the rate and permutates the result.

obtain a longer digest, we can continue to permute and read from the rate part of the state as figure 2.16 shows.

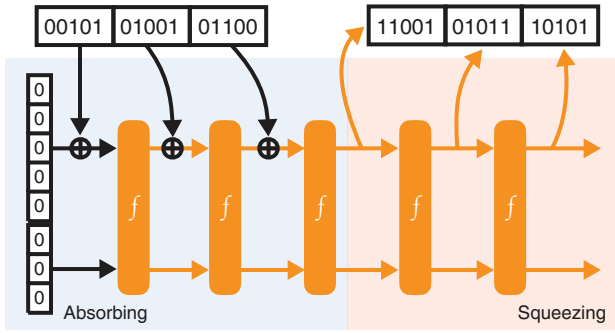

Figure 2.16 To obtain a digest with the sponge construction, one iteratively permutes the state and retrieves as much rate (the upper part of the state) as needed.

And this is how SHA-3 works. Because it is a *sponge construction*, ingesting the input is naturally called *absorbing* and creating the digest is called *squeezing*. The sponge is specified with a 1,600-bit permutation using different values for r and c, depending on the security advertised by the different versions of SHA-3.

SHA-3 is a random oracle

I talked about random oracles earlier: an ideal and fictional construction that returns perfectly random responses to queries and repeats itself if we query it with the same input twice. It turns out that the sponge construction behaves closely to a random oracle, as long as the permutation used by the construction looks random enough. How do we prove such security properties on the permutation? Our best approach is to try to break it, many times, until we gain strong confidence in its design (which is what happened during the SHA-3 competition). The fact that SHA-3 can be modeled as a random oracle instantly gives it the security properties we would expect from a hash function.

2.5.3 SHAKE and cSHAKE: Two extendable output functions (XOF)

I introduced the two major hash function standards: SHA-2 and SHA-3. These are well-defined hash functions that take arbitrary-length inputs and produce random-looking and fixed-length outputs. As you will see in later chapters, cryptographic protocols often necessitate this type of primitives but do not want to be constrained by the fixed sizes of a hash function's digest. For this reason, the SHA-3 standard introduced a more versatile primitive called an *extendable output function* or *XOF* (pronounced "zoff"). This section introduces the two standardized XOFs: SHAKE and cSHAKE.

SHAKE, specified in FIPS 202 along with SHA-3, can be seen as a hash function that returns an output of an arbitrary length. SHAKE is fundamentally the same construction as SHA-3, except that it is faster and permutes as much as you want it to permute in the squeezing phase. Producing outputs of different sizes is quite useful, not only to create a digest, but also to create random numbers, to derive keys, and so on. I will talk about the different applications of SHAKE again in this book; for now, imagine that SHAKE is like SHA-3 except that it provides an output of any length you might want.

This construction is so useful in cryptography that one year after SHA-3 was standardized, NIST published its Special Publication 800-185 containing a *customizable SHAKE* called *cSHAKE*. cSHAKE is pretty much exactly like SHAKE, except that it also takes a customization string. This customization string can be empty, or it can be any string you want. Let's first see an example of using cSHAKE in pseudocode:

```
cSHAKE(input="hello world", output_length=256, custom_string="my_hash")
-> 72444fde79690f0cac19e866d7e6505c
cSHAKE(input="hello world", output_length=256, custom_string="your_hash")
-> 688a49e8c2a1e1ab4e78f887c1c73957
```

As you can see, the two digests differ even though cSHAKE is as deterministic as SHAKE and SHA-3. This is because a different customization string was used. A *customization string* allows you to customize your XOF! This is useful in some protocols where, for example, different hash functions must be used in order to make a proof work. We call this *domain separation*.

As a golden rule in cryptography: if the same cryptographic primitive is used in different use cases, do not use it with the same key (if it takes a key) or/and apply domain separation. You will see more examples of domain separation as we survey cryptographic protocols in later chapters.

> **WARNING** NIST tends to specify algorithms that take parameters in bits instead of bytes. In the example, a length of 256 bits was requested. Imagine if you had requested a length of 16 bytes and got 2 bytes instead, due to the program thinking you had requested 16 bits of output. This issue is sometimes called a *bit attack*.

As with everything in cryptography, the length of cryptographic strings like keys, parameters, and outputs is strongly tied to the security of the system. It is important

that one does not request too short outputs from SHAKE or cSHAKE. *One can never go wrong by using an output of 256 bits* as it provides 128 bits of security against collision attacks. But real-world cryptography sometimes operates in constrained environments that could use shorter cryptographic values. This can be done if the security of the system is carefully analyzed. For example, if collision resistance does not matter in the protocol making use of the value, pre-image resistance only needs 128-bit long outputs from SHAKE or cSHAKE.

2.5.4 Avoid ambiguous hashing with TupleHash

In this chapter, I have talked about different types of cryptographic primitives and cryptographic algorithms. This included

- The SHA-2 hash function, which is vulnerable to length-extension attacks but still widely used when no secrets are hashed
- The SHA-3 hash function, which is the recommended hash function nowadays
- The SHAKE and cSHAKE XOFs, which are more versatile tools than hash functions because they offer a variable output length

I will talk about one more handy function, *TupleHash*, which is based on cSHAKE and specified in the same standard as cSHAKE. TupleHash is an interesting function that allows one to hash a *tuple* (a list of something). To explain what TupleHash is and why it is useful, let me tell you a story.

A few years ago I was tasked to review a cryptocurrency as part of my work. It included basic features one would expect from a cryptocurrency: accounts, payments, and so on. Transactions between users would contain metadata about who is sending how much to whom. It would also include a small fee to compensate the network for processing the transaction.

Alice, for example, can send transactions to the network, but to have them accepted, she needs to include proof that the transaction came from her. For this, she can hash the transaction and sign it (I gave a similar example in chapter 1). Anyone can hash the transaction and verify the signature on the hash to see that this is the transaction Alice meant to send. Figure 2.17 illustrates that a man-in-the-middle (MITM) attacker who intercepts the transaction before it reaches the network would not be

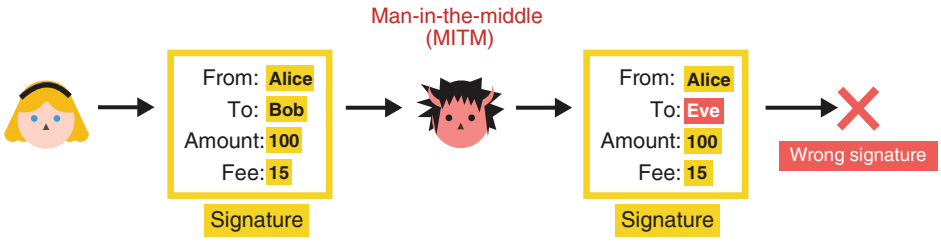

Figure 2.17 Alice sends a transaction as well as a signature over the hash of the transaction. If a MITM attacker attempts to tamper with the transaction, the hash will be different and, thus, the attached signature will be incorrect.

able to tamper with the transaction. This is because the hash would change, and the signature would then not verify the new transaction digest.

You will see in chapter 7 that such an attacker is, of course, unable to forge Alice's signature on a new digest. And thanks to the second pre-image resistance of the hash function used, the attacker cannot find a totally different transaction that would hash to the same digest either.

Is our MITM attacker harmless? We're not out of the woods yet. Unfortunately, for the cryptocurrency I was auditing, the transaction was hashed by simply concatenating each field:

```
$ echo -n "Alice""Bob""100""15" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

What appeared as totally fine, actually completely broke the cryptocurrency's payment system. Doing this trivially allows an attacker to break the second pre-image resistance of the hash function. Take a few moments to think about how you could find a different transaction that hashes to the same digest, `34d6...`.

What happens if we move one digit from the *fee* field to the *amount* field? One can see that the following transaction hashes to the same digest Alice signed:

```
$ echo -n "Alice""Bob""1001""5" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

And thus, a MITM attacker who would want Bob to receive a bit more money would be able to modify the transaction without invalidating the signature. As you've probably guessed, this is what TupleHash solves. It allows you to unambiguously hash a list of fields by using non-ambiguous encoding. What happens in reality is something close to the following (with the || string concatenation operation):

```
cSHAKE(input="5"||"Alice"||"3"||"Bob"||"3"||"100"||"2"||"10",
  ➥ output_length=256, custom_string="TupleHash"+"anything you want")
```

The input is this time constructed by prefixing each field of the transaction with its length. Take a minute to understand why this solves our issue. In general, one can use any hash function safely by always making sure to *serialize* the input before hashing it. Serializing the input means that there always exists a way to *deserialize* it (meaning to recover the original input). If one can deserialize the data, then there isn't any ambiguity on field delimitation.

2.6 Hashing passwords

You have seen several useful functions in this chapter that either are hash functions or extend hash functions. But before you can jump to the next chapter, I need to mention *password hashing*.

Imagine the following scenario: you have a website (which would make you a webmaster) and you want to have your users register and log in to the site, so you create

two web pages for these two respective features. Suddenly, you wonder, how are you going to store their passwords? Do you store those in cleartext in a database? There seems to be nothing wrong with this at first, you think. It is not perfect though. People tend to reuse the same password everywhere and if (or when) you get breached and attackers manage to dump all of your users' passwords, it will be bad for your users, and it will be bad for the reputation of your platform. You think a little bit more, and you realize that an attacker who would be able to steal this database would then be able to log in as any user. Storing the passwords in cleartext is now less than ideal and you would like to have a better way to deal with this.

One solution could be to hash your passwords and only store the digests. When someone logs in to your website, the flow would be similar to the following:

1. You receive the user's password.
2. You hash the password they give you and get rid of the password.
3. You compare the digest with what you had stored previously; if it matches, the user is logged in.

The flow allows you to handle users' passwords for a limited time. Still, an attacker that gets into your servers can stealthily remain to log passwords from this flow until you detect its presence. We acknowledge that this is still not a perfect situation, but we still improved the site's security. In security, we also call this *defense in depth*, which is the act of layering imperfect defenses in hope that an attacker will not defeat all of those layers. This is what real-world cryptography is also about. But other problems exist with this solution:

- *If an attacker retrieves hashed passwords, a brute force attack or an exhaustive search (trying all possible passwords) can be undertaken.* This would test each attempt against the whole database. Ideally, we would want an attacker to only be able to attack one hashed password at a time.
- *Hash functions are supposed to be as fast.* Attackers can leverage this to brute force (many, many passwords per second). Ideally, we would have a mechanism to slow down such attacks.

The first issue has been commonly solved by using *salts*, which are random values that are public and different for each user. We use a salt along with the user's password when hashing it, which in some sense is like using a per-user customization string with cSHAKE: it effectively creates a different hash function for every user. Because each user uses a different hash function, an attacker cannot precompute large tables of passwords (called *rainbow tables*), hoping to test those against the whole database of stolen password hashes.

The second issue is solved with *password hashes*, which are designed to be slow. The current state-of-the-art choice for this is *Argon2*, the winner of the Password Hashing Competition (https://password-hashing.net) that ran from 2013 to 2015. At the time of this writing (2021), Argon2 is on track to be standardized as an RFC (https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/). In practice, other nonstandard algorithms like

PBKDF2, bcrypt, and scrypt are also used. The problem is that these can be used with insecure parameters and are, thus, not straightforward to configure in practice.

In addition, only Argon2 and scrypt defend against heavy optimizations from attackers as other schemes are not memory hard. The term *memory hard* means that the algorithm can only be optimized through the optimization of memory access. In other words, optimizing the rest doesn't gain you much. As optimizing memory access is limited even with dedicated hardware (there's only so much cache you can put around a CPU), memory-hard functions are slow to run on any type of device. This is a desired property when you want to prevent attackers from getting a non-negligible speed advantage in evaluating a function.

Figure 2.18 reviews the different types of hash functions you saw in this chapter.

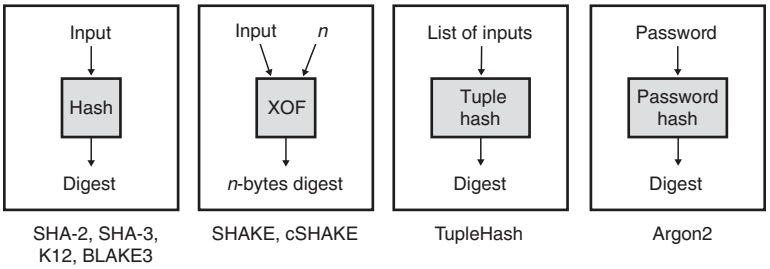

Figure 2.18 In this chapter, you saw four types of hash functions: (1) the normal kind that provide a unique random-looking identifier for arbitrary-length inputs; (2) extendable output functions that are similar but provide an arbitrary-length output; (3) tuple hash functions that unambiguously list hash values; and (4) password-hashing functions that can't be easily optimized in order to store passwords safely.

Summary

- A hash function provides collision resistance, pre-image resistance, and second pre-image resistance.
  - Pre-image resistance means that one shouldn't be able to find the input that produced a digest.
  - Second pre-image resistance means that from an input and its digest, one shouldn't be able to find a different input that hashes to the same digest.
  - Collision resistance means that one shouldn't be able to find two random inputs that hash to the same output.
- The most widely adopted hash function is SHA-2, while the recommended hash function is SHA-3 due to SHA-2's lack of resistance to length-extension attacks.
- SHAKE is an extendable output function (XOF) that acts like a hash function but provides an arbitrary-length digest.

- cSHAKE (for customizable SHAKE) allows one to easily create instances of SHAKE that behave like different XOFs. This is called domain separation.
- Objects should be serialized before being hashed in order to avoid breaking the second pre-image resistance of the hash function. Algorithms like TupleHash automatically take care of this.
- Hashing passwords make use of slower hash functions designed specifically for that purpose. Argon2 is the state-of-the-art choice.